ECE/CS 552: Introduction to Computer Architecture
ASSIGNMENT #3
Due Date: At the beginning of class, March 6th

<u>This homework is to be done individually.</u>

1. (20 pts.) In this exercise, we examine how data dependencies affect execution in the basic five-stage pipeline described in the textbook. Problems in this exercise refer to the following sequence of instructions:

a.	lw $1, 40 ($6) add $6, $2, $2 sw $6, 10($3)
b.	lw $5, -16 ($5) sw $5, -16 ($5) add $5, $5, $5

1. (3 pts.) Indicate dependences and their type.
2. (4 pts.) Assume there is no forwarding in this pipelined processor. Indicate hazards and add "nop" instructions to eliminate them.
3. (4 pts.)Assume there is full forwarding. Indicate hazards and add "nop" instructions to eliminate them,

   The remaining problems in this exercise assume the following clock cycle times:

Without forwarding	With full forwarding	With ALU-ALU forwarding only
300 ps	400 ps	360 ps

4. (3 pts)What is the total execution time of this instruction sequence without forwarding and with full forwarding? What is the speed-up achieved by adding full forwarding to a pipeline that had no forwarding?
5. (3 pts.)Add "nop" instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to EX stage).
6. (3 pts.)What is the total execution time of this instruction sequence with only ALU_ALU forwarding? What is the speed-up over a no-forwarding pipeline?

2. (9 pts.) This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes:

	Branch Outcomes
a.	T, T, NT, T
b.	T, T, T, NT, NT

a) (3 pts.)What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?
b) (3 pts.)What is the accuracy of the two-bit predictor for the first four branches in these patterns, assuming that the predictor starts off in the bottom left state (predict not taken) from Figure 1.

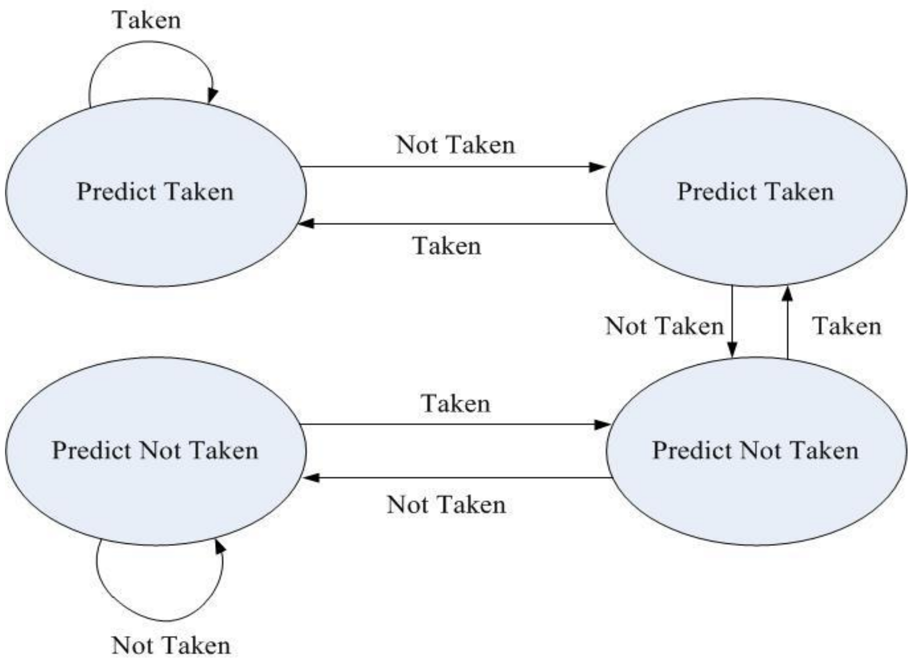

Figure 1: State Machine of 2-bit branch predictor

c) (3 pts.)What is the accuracy of the two-bit predictor if this pattern is repeated forever?

3. (18 pts.) Consider the ISA you have to implement for the project.

   i. (1 pt) Circle the pipeline mechanism used to handle a taken branch

   Bypassing **Flush** **Stall**

   ii. (1.5 pts) How many clock cycles are "lost" (wasted) on a taken branch?

   iii. (1.5 pts) How many clock cycles are "lost" on a not taken branch?

   iv. (1.5 pts) How many clock cycles are lost on:

   LW R2, R1, 0x00
   SW R2, R1, 0x01

   v. (1.5 pts) How many clock cycles are lost on:

   LW R1, R1, 0x00
   ADD R3, R2, R1

   vi. (4 pts) Write a short paragraph explaining what delayed branches are, and how they can help CPI

   vii. (4 pts) The ISA contains an ADDZ instruction. This is a conditionally executing instruction. Name another processor architecture that makes extensive use of conditionally executing instructions. Explain how a conditionally executing instructions can help CPI

   viii. (3 pts) What is the load/use penalty for our ISA and the 5-stage pipeline we intend to implement? Circle the pipline mechanism used to solve load use hazards.

   Detech & Bypass **Detect & Flush** **Detect & Stall**

4. (13 pts) Test writing for our WISC-S14 ISA. One great thing about Icarus Verilog is that the compilation step produces a platform independent model that can be executed by **vvp**. This allows us to post a compiled model of a processor that executes our ISA so you can write tests against it. If you are a MAC user you should already have Icarus Verilog installed, thus have **vvp**. If you are a windows user download the installer package posted in the HW3 portion of the webpage and install Icarus Verilog.

   Now download the WISC.S14 from the HW3 portion of the project webpage.

   Our assembler for the project is written in perl. So you will need a perl interpreter installed on your computer. I use ActivePerl. Download a perl interpreter on your machine so you can run our assembler.

   Download our assembler **asmbl.pl**

   Now you should be able to write programs in assembly and use **assmbl.pl** to generate machine language.

   The executable model looks for its machine code in a file called **instr.hex** that resides in the same directory that the model (WISC.S14) resides in.

   Write a self-checking test to test 4 instructions from the WISC-S14 ISA (4 –instructions other than LLB, SUB, B EQ, comment what you are trying to test in the test. # is the comment character for assembly). Use the self-checking method that was outlined and practiced in class on Tues Feb 25th. Use the assembler to generate a **instr.hex** file.

   ***Command_prompt*>perl asmbl.pl myTest.asm > instr.hex**

   Now run the generate **instr.hex** file against the WISC.S14 model.

   ***Command_prompt*>vvp WISC.S14**

   Submit a print out of your assembly test, and a screen shot of the **vvp** output to show you ran it.

5. (40 pts.) Single cycle non-branching execution engine (working you ½ way toward a single cycle implementation of our ISA).
Using your ALU from HW2, build the CPU outlined in the figure below. You will need to implement several new blocks (Simple program counter that resets, increments, and can be halted; a source mux that can select RF or SE immediate results; and finally an instruction decode block). **The Verilog for the instruction memory (IM) and register file are provided** through the course website. This execution engine should be able to execute any of the instructions listed in the table below:

ADD	ADDZ	SUB	AND	NOR	SLL	SRL	SRA	LLB	LHB	HLT

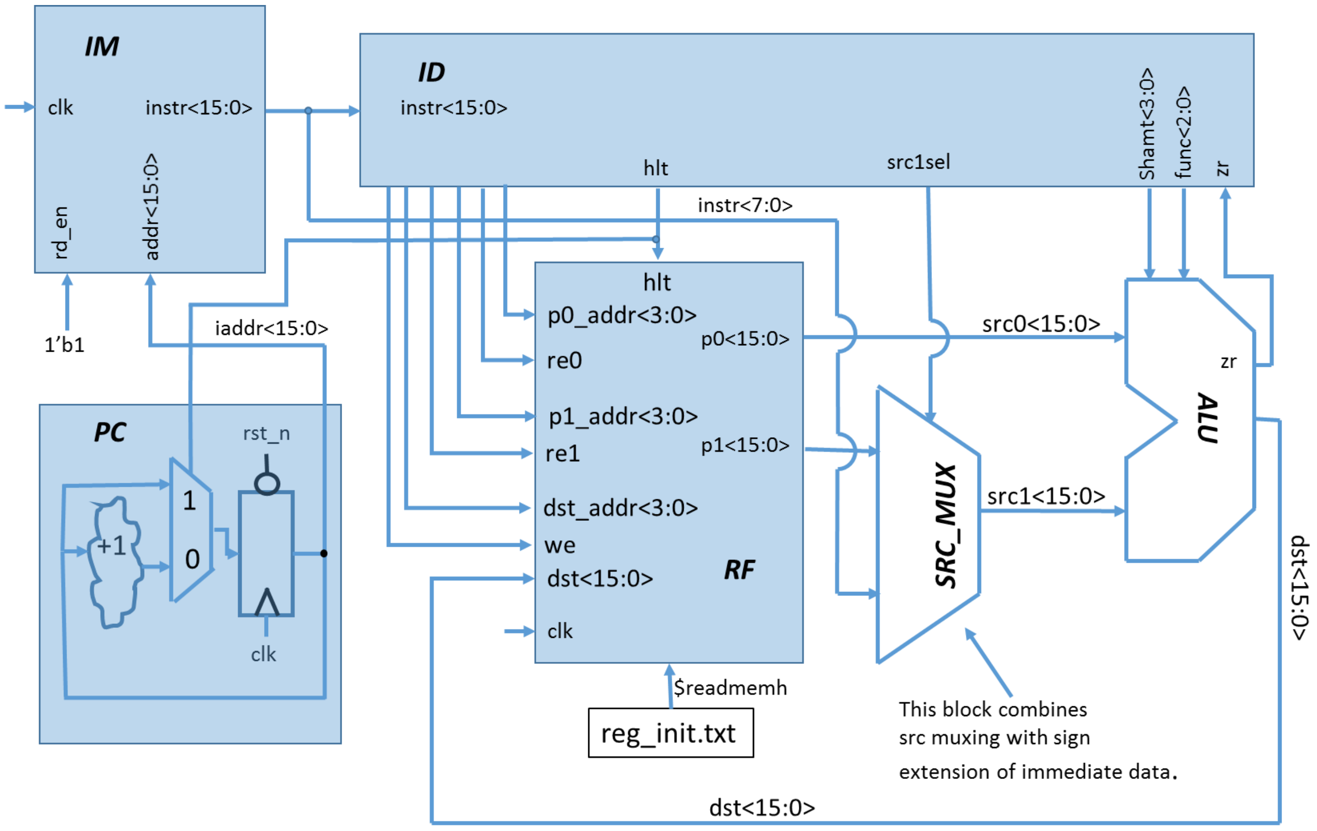

Figure 2: Block diagram of single-cycle non-branching execution engine

A file called ***reg_init.txt*** is available on the course website. This file is used by the register file model to initialize the registers with values. Register 0 will contain 0x0000, register 1 will contain 0x1111, register 2 will contain 0x2222, … Note that this is not realistic. In real life a register file is not initialized at power up and registers contain unknown values. This was just done to ease testing.

Implement this execution engine in Verilog. Generally follow the naming of signals outlined here. **Submit Verilog** for the SRC_MUX, the PC, ID, Your top level that ties them together (cpu.v), and for your testbench (cpu_tb.v).

Also note that the instruction decode block will assert a signal called hlt when it encounters the HLT instruction. This signal serves two purposes. It halts the program counter, and it causes the RF to dump all register contents to standard out.

The top level signals of your non-branching execution engine (in a file called cpu.v) should be as specified in the table below:

cpu.v interface		
Signal:	**Direction:**	**Description:**
clk	input	System clock provided by test bench
rst_n	input	Active low asynchronous reset input, provided by test bench
hlt	output	Asserted by design when the HLT instruction is encountered. Perhaps not used by your test bench now, but it will be later.

For the testing you will write non-branching streams of instructions in assembly and use the provided assembler (asmbl.pl) to generate machine code that is loaded (via a $readmemh) into the instruction memory. **Note** that you may have to modify the Verilog for the instruction memory to point to the full path of the output of the assembler for your tests.

Submit the assembly code file(s) you used to test your design.